Introduction to parallel programming using MPI

CPPG tutorial

December 15, 2017

Stéphane Ethier

(ethier@pppl.gov)

Computational Plasma Physics Group

Why Parallel Computing?

Why not run *n* instances of my code *à la* MapReduce/Hadoop?

- Want to speed up your calculation
- Your problem size is too large for a single node
- Want to use those extra cores on your multicore processor
- Solution:
  - Split the work between several processor cores so that they can work in parallel
  - Exchange data between them when needed
- How?
  - OpenMP directives on shared memory node
  - Message Passing Interface (MPI) on distributed memory systems (works also on shared memory nodes!)
  - and others (Fortran Co-Arrays, OpenSHMEM, UPC, ...)

What is MPI?

- MPI stands for Message Passing Interface
- It is a message-passing specification, a standard for the vendors to implement
- In practice, MPI is a library consisting of C functions and Fortran subroutines (Fortran) used for exchanging data between processes
- An MPI library exists on **ALL** parallel computers so it is **highly portable**
- The scalability of MPI is not limited by the number of processors/cores on one computation node, as opposed to shared memory parallel models
- Also available for Python (mpi4py.scipy.org), R (Rmpi), Lua, and Julia! (if you can call C functions, you can use MPI...)

MPI standard

- The MPI standard is a specification of what MPI is and how it should behave. Vendors have some flexibility in the implementation (e.g. buffering, collectives, topology optimizations, etc.).
- This tutorial focuses on the functionality introduced in the original MPI-1 standard
- MPI-2 standard introduced additional support for
  - Parallel I/O (many processes writing to a single file). Requires a parallel filesystem to be efficient
  - One-sided communication: MPI_Put, MPI_Get
  - Dynamic Process Management
- MPI-3 standard starting to be implemented by compilers vendors
  - Non-blocking collectives
  - Improved one-sided communications
  - Improved Fortran bindings for type check
  - And more (see http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf)

Why do I need to know both MPI?

#	Site	Manufacturer	Computer	Country	Cores	Rmax [Pflops]	Power [MW]
1	National Supercomputing Center in Wuxi	NRCPC	Sunway TaihuLight NRCPC Sunway SW26010, 260C 1.45GHz	China	10,649,600	93.0	15.4
2	National University of Defense Technology	NUDT	Tianhe-2 NUDT TH-IVB-FEP, Xeon 12C 2.2GHz, IntelXeon Phi	China	3,120,000	33.9	17.8
3	Swiss National Supercomputing Centre (CSCS)	Cray	Piz Daint Cray XC50, Xeon E5 12C 2.6GHz, Aries, NVIDIA Tesla P100	Switzerland	361,760	19.6	2.27
4	Japan Agency for Marine-Earth Science and Technology	ExaScaler	Gyoukou ZettaScaler-2.2 HPC System, Xeon 16C 1.3GHz, IB-EDR, PEZY-SC2 700Mhz	Japan	19,860,000	19.1	1.35
5	Oak Ridge National Laboratory	Cray	Titan Cray XK7, Opteron 16C 2.2GHz, Gemini, NVIDIA K20x	USA	560,640	17.6	8.21
6	Lawrence Livermore National Laboratory	IBM	Sequoia BlueGene/Q, Power BQC 16C 1.6GHz, Custom	USA	1,572,864	17.2	7.89
7	Los Alamos NL / Sandia NL	Cray	Trinity Cray XC40, Intel Xeon Phi 7250 68C 1.4GHz, Aries	USA	979,968	14.1	3.84
8	Lawrence Berkeley National Laboratory	Cray	Cori Cray XC40, Intel Xeons Phi 7250 68C 1.4 GHz, Aries	USA	622,336	14.0	3.94
9	JCAHPC Joint Center for Advanced HPC	Fujitsu	Oakforest-PACS PRIMERGY CX1640 M1, Intel Xeons Phi 7250 68C 1.4 GHz, OmniPath	Japan	556,104	13.6	2.72
10	RIKEN Advanced Institute for Computational Science	Fujitsu	K Computer SPARC64 VIIIfx 2.0GHz, Tofu Interconnect	Japan	795,024	10.5	12.7

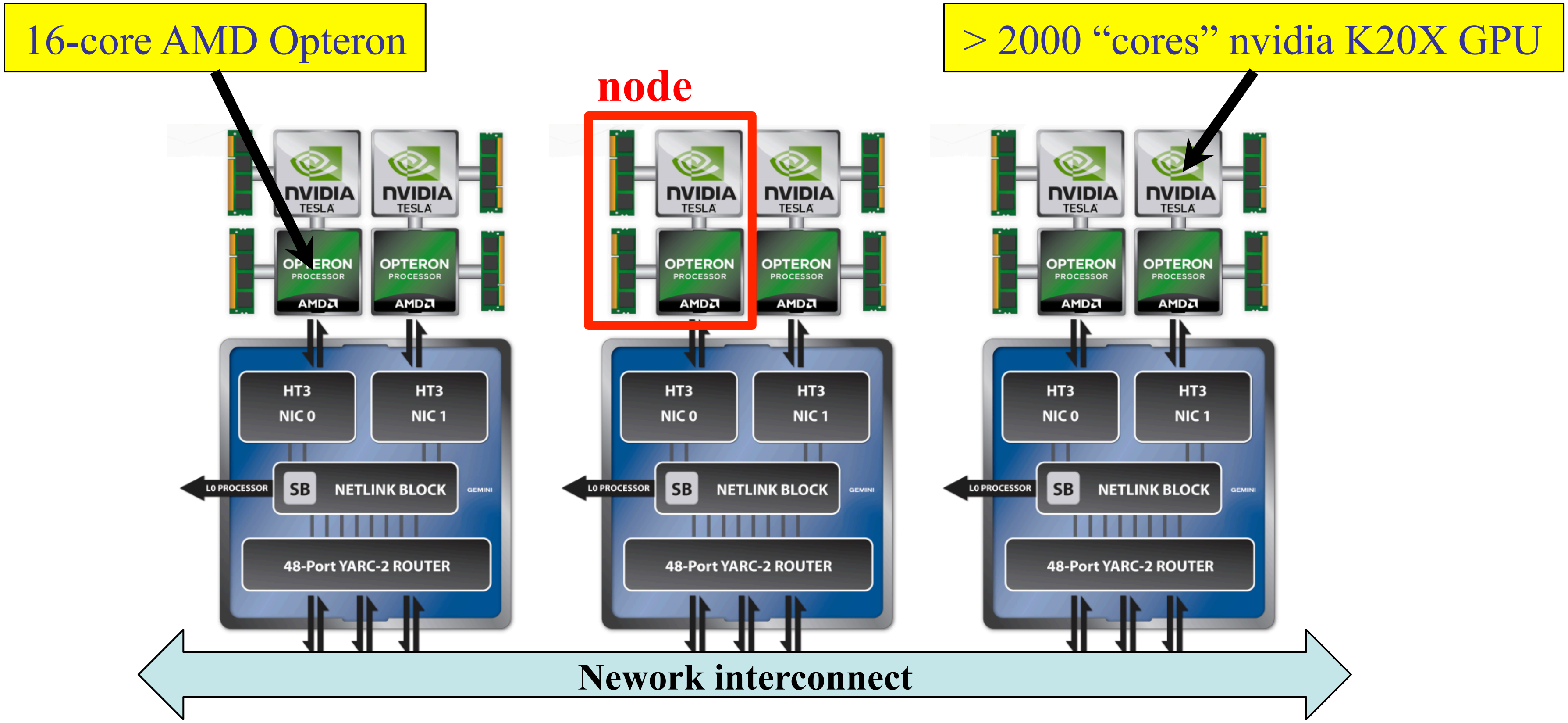
List of top supercomputers in the world

(www.top500.org)

Titan Cray XK7 hybrid system at OLCF

Processor: AMD Interlagos (16)	GPUs: 18,688 Tesla K20
Cabinets: 200	Memory/node CPU: 32 GB
# nodes: 18,688	Memory/node GPU: 6 GB
# cores/node: 16	Interconnect: Gemini
Total cores: 299,008	Speed: 27 PF peak (17.6)

Cray XK7 architecture



MPI

Context: Distributed memory parallel computers

- Each processor has its own memory and cannot access the memory of other processors
- A copy of the same executable runs on each MPI process (processor core)
- Any data to be shared must be explicitly transmitted from one to another

Most message passing programs use the *single program multiple data* (SPMD) model

- Each processor executes the same set of instructions
- Parallelization is achieved by letting each processor operate on a different piece of data
- Not to be confused with SIMD: Single Instruction Multiple Data *a.k.a vector computing*

A sample MPI program in Fortran 90

```
Program mpi_code
   ! Load MPI definitions
     use mpi  (or include mpif.h)

   ! Initialize MPI
     call MPI_Init(ierr)
   ! Get the number of processes
     call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
   ! Get my process number (rank)
     call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

     Do work and make message passing calls…

   ! Finalize
     call MPI_Finalize(ierr)

end program mpi_code
```

Header file

```
Program mpi_code
   ! Load MPI definitions
     use mpi

   ! Initialize MPI
     call MPI_Init(ierr)
   ! Get the number of processes
     call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
   ! Get my process number (rank)
     call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

     Do work and make message passing calls…

   ! Finalize
     call MPI_Finalize(ierr)

end program mpi_code
```

- Defines MPI-related parameters and functions
- Must be included in all routines calling MPI functions
- Can also use include file:
  include mpif.h

Initialization

```
Program mpi_code
   ! Load MPI definitions
     use mpi

   ! Initialize MPI
     call MPI_Init(ierr)
   ! Get the number of processes
     call MPI_Comm_size(MPI_COMM_WO
   ! Get my process number (rank)
     call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

     Do work and make message passing calls…

   ! Finalize
     call MPI_Finalize(ierr)

end program mpi_code
```

- Must be called at the beginning of the code before any other calls to MPI functions
- Sets up the communication channels between the processes and gives each one a rank.

How many processes do we have?

- Returns the number of processes available under MPI_COMM_WORLD communicator
- This is the number used on the mpiexec (or mpirun) command:

  mpiexec –n nproc a.out

```
  call MPI_Init(ierr)
! Get the number of processes
  call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
! Get my process number (rank)
  call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

  Do work and make message passing calls…

! Finalize
  call MPI_Finalize(ierr)

end program mpi_code
```

What is my rank?

```
Program mpi_code
  ! Load MPI definitions
```

- Get my rank among all of the nproc processes under MPI_COMM_WORLD
- This is a unique number that can be used to distinguish this process from the others

```
    call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
  ! Get my process number (rank)
    call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

    Do work and make message passing calls…

  ! Finalize
    call MPI_Finalize(ierr)

end program mpi_code
```

Termination

```
Program mpi_code
   ! Load MPI definitions
     use mpi (or include mpif.h)

   ! Initialize MPI
     call MPI_Init(ierr)
   ! Get the number of processes
     call MPI_Comm_size(MPI_COMM_WORLD,nproc,ierr)
   ! Get my process number (rank)
     call MPI_Comm_rank(MPI_COMM_WORLD,myrank,ierr)

     Do work and make message passing calls…

   ! Finalize
     call MPI_Finalize(ierr)

end program mpi_code
```

- Must be called at the end of the properly close all communication channels
- No more MPI calls after finalize

A sample MPI program in C

```
#include "mpi.h"
int main( int argc, char *argv[] )
{
  int nproc, myrank;
  /* Initialize MPI */
    MPI_Init(&argc,&argv);
  /* Get the number of processes */
    MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  /* Get my process number (rank) */
    MPI_Comm_rank(MPI_COMM_WORLD,&myrank);

    Do work and make message passing calls…

  /* Finalize */
    MPI_Finalize();
 return 0;
}
```

How much do I need to know?

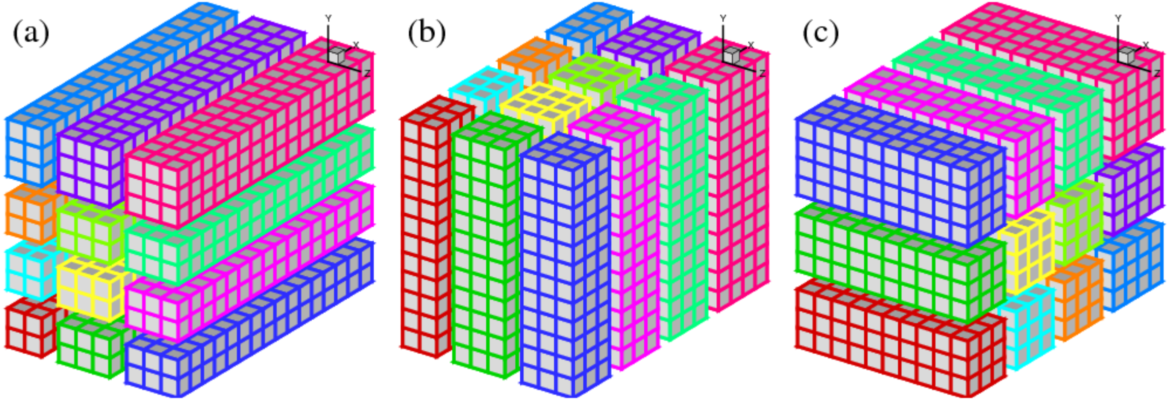
- MPI-1 has over 125 functions/subroutines
- Can actually do everything with about 6 of them although I would not recommend it
- Collective functions are EXTREMELY useful since they simplify the coding and vendors optimize them for their interconnect hardware
- One can access flexibility when it is required.
- One need not master all parts of MPI to use it.

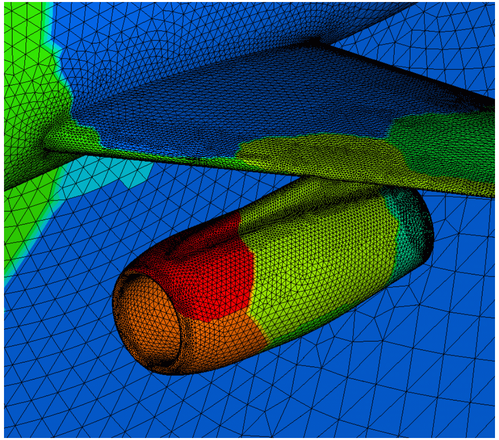
MPI Communicators

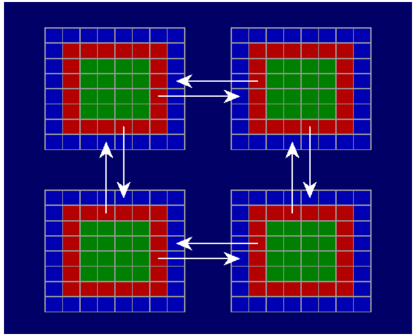
- A communicator is an identifier associated with a group of processes
  - Each process has a unique rank within a specific communicator (the rank starts from 0 and has a maximum value of (nprocesses-1)).
  - Internal mapping of processes to processing units
  - Always required when initiating a communication by calling an MPI function or routine.
- Default communicator MPI_COMM_WORLD, which contains all available processes.
- Several communicators can coexist
  - A process can belong to different communicators at the same time, but has a unique rank in each communicator

Okay... but how do we split the work between ranks?
Domain Decomposition!

- Most widely used method for grid-based calculations

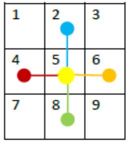






How to split the work between ranks?
Split matrix elements in PDE solves

- See PETSc project: https://www.mcs.anl.gov/petsc/



(a)

Grid Element	1		2		3		4		5		6		7		8		9	
1	57	58	93	94			125	126										
	59	60	95	96			127	128										
2	25	26	61	62	97	98			129	130								
	27	28	63	64	99	100			131	132								
3			29	30	65	66	101	102			133	134						
			31	32	67	68	103	104			135	136						
4	1	2			33	34	69	70	105	106			137	138				
	3	4			35	36	71	72	107	108			139	140				
5			5	6			37	38	73	74	109	110			141	142		
			7	8			39	40	75	76	111	112			143	144		
6					9	10			41	42	77	78	113	114			145	146
					11	12			43	44	79	80	115	116			147	148
7							13	14			45	46	81	82	117	118		
							15	16			47	48	83	84	119	120		
8									17	18			49	50	85	86	121	122
									19	20			51	52	87	88	123	124
9											21	22			53	54	89	90
											23	24			55	56	91	92

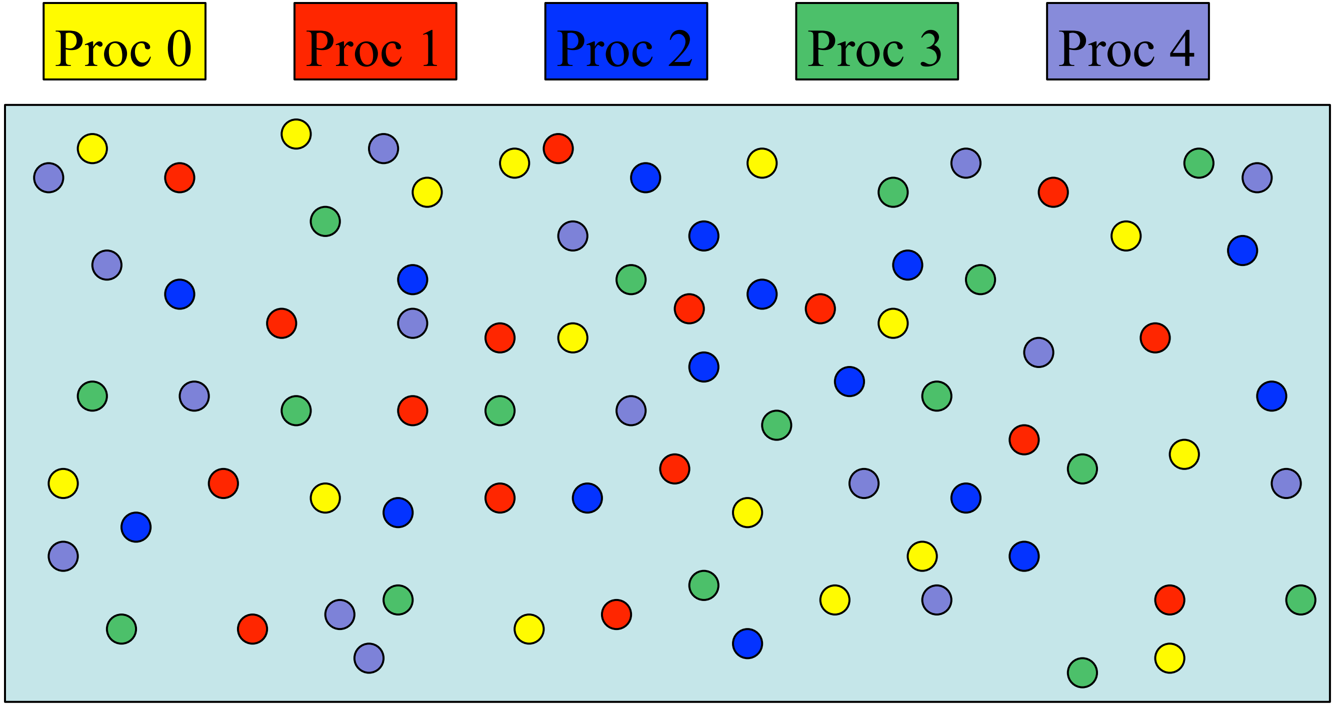
(b)

-3		-1		0		1		3	
00	00	00	00	57	58	93	94	125	126
00	00	00	00	59	60	95	96	127	128
00	00	25	26	61	62	97	98	129	130
00	00	27	28	63	64	99	100	131	132
00	00	29	30	65	66	101	102	133	134
00	00	31	32	67	68	103	104	135	136
1	2	33	34	69	70	105	106	137	138
3	4	35	36	71	72	107	108	139	140
5	6	37	38	73	74	109	110	141	142
7	8	39	40	75	76	111	112	143	144
9	10	41	42	77	78	113	114	145	146
11	12	43	44	79	80	115	116	147	148
13	14	45	46	81	82	117	118	00	00
15	16	47	48	83	84	119	120	00	00
17	18	49	50	85	86	121	122	00	00
19	20	51	52	87	88	123	124	00	00
21	22	53	54	89	90	00	00	00	00
23	24	55	56	91	92	00	00	00	00

(c)

How to split the work between ranks?
"Coloring"

- Useful for particle simulations



Compiling and linking an MPI code

- Need to tell the compiler where to find the MPI include files and how to link to the MPI libraries.
- Fortunately, most MPI implementations come with scripts that take care of these issues:
  - mpicc mpi_code.c –o a.out
  - mpiCC mpi_code_C++.C –o a.out
  - mpif90 mpi_code.f90 –o a.out
- Two widely used (and free) MPI implementations on Linux clusters are:
  - MPICH (http://www-unix.mcs.anl.gov/mpi/mpich)
  - OPENMPI (http://www.openmpi.org)

Makefile

- Always a good idea to have a Makefile

```
%cat Makefile
CC=mpicc
CFLAGS=-O

% : %.c
	$(CC) $(CFLAGS) $< -o $@
```

How to run an MPI executable

- The implementation supplies scripts to launch the MPI parallel calculation, for example:

```
mpirun -np nproc a.out
mpiexec -n nproc a.out
```

MPICH, OPENMPI

```
aprun -size nproc a.out (Cray XT)
srun -n nproc a.out (SLURM batch system)
```

- A copy of the same program runs on each processor core within its own process (private address space).
- Each process works on a subset of the problem.
- Exchange data when needed
  - Can be exchanged through the network interconnect
  - Or through the shared memory on SMP machines (Bus?)
- Easy to do coarse grain parallelism = <u>scalable</u>

mpirun and mpiexec

- Both are used for starting an MPI job
- If you don't have a batch system, use mpirun

```
mpirun -np #proc -hostfile mfile a.out >& out < in &
```

```
%cat mfile
 machine1.princeton.edu                  machine1.princeton.edu
 machine2.princeton.edu      OR          machine1.princeton.edu
 machine3.princeton.edu                  machine1.princeton.edu
 machine4.princeton.edu                  machine1.princeton.edu
```

1 MPI process per host

4 MPI processes on same host

- SLURM batch system takes care of assigning the hosts

Batch System

- Submit a job script: sbatch script
- Check status of jobs: squeue –a   (for all jobs)
- Stop a job: scancel job_id

```
#!/bin/bash
#SBATCH --job-name=test
#SBATCH --partition=dawson # partition (dawson, ellis or kruskal)
#SBATCH -N 1 # number of nodes
#SBATCH -n 1 # number of cores
#SBATCH --mem 100 # memory to be used per node
#SBATCH -t 0-2:00 # time (D-HH:MM)
#SBATCH -o slurm.%N.%j.out # STDOUT
#SBATCH -e slurm.%N.%j.err # STDERR
#SBATCH --mail-type=END,FAIL # notifications for job done & fail
#SBATCH --mail-user=myemail@pppl.gov # send-to address
module load gcc/6.1.0
module load openmpi/1.10.3
mpiexec ./mpihello
```

Basic MPI calls to exchange data

- Point-to-Point communications
  - Only 2 processes exchange data
  - It is the basic operation of all MPI calls
- Collective communications
  - A single call handles the communication between all the processes in a communicator
  - There are 3 types of collective communications
    - Data movement (e.g. MPI_Bcast)
    - Reduction (e.g. MPI_Reduce)
    - Synchronization: MPI_Barrier

Point-to-point communication

<u>Point to point:</u> **2 processes at a time**

```
MPI_Send(buf,count,datatype,dest,tag,comm,ierr)

MPI_Recv(buf,count,datatype,source,tag,comm,status,ierr)
```

```
MPI_Sendrecv(sendbuf,sendcount,sendtype,dest,sendtag,
     recvbuf,recvcount,recvtype,source,recvtag,comm,status,ierr)
```

where the datatypes are:

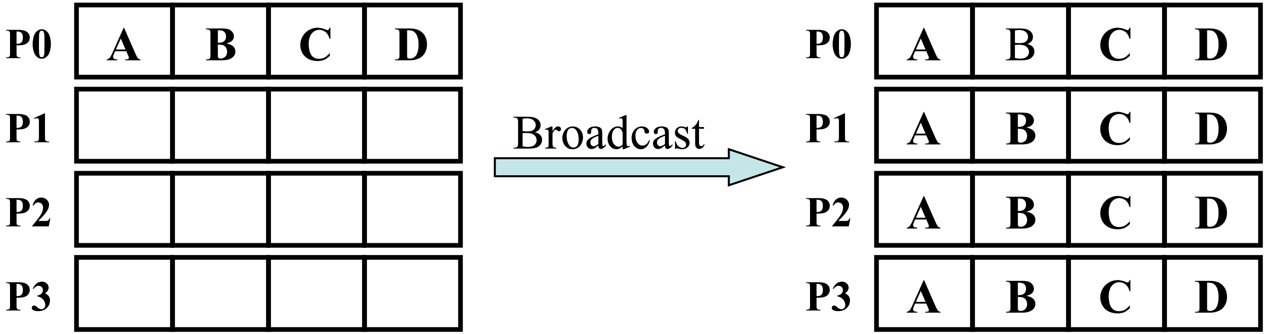
FORTRAN: `MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX,MPI_CHARACTER, MPI_LOGICAL, etc...`

C : `MPI_INT, MPI_LONG, MPI_SHORT, MPI_FLOAT, MPI_DOUBLE, etc...`

Predefined Communicator: `MPI_COMM_WORLD`

Collective communication: Broadcast

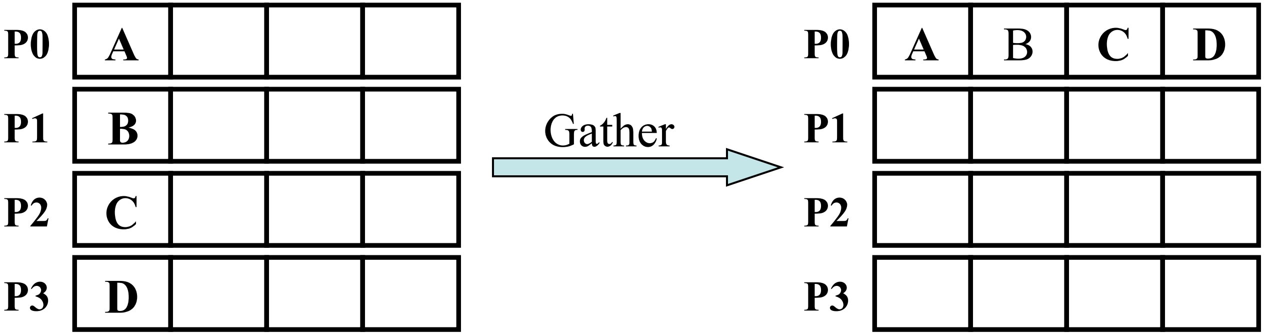
```
MPI_Bcast(buffer,count,datatype,root,comm,ierr)
```



- One process (called “root”) sends data to all the other processes in the same communicator
- Must be called by <u>ALL</u> processes with the same arguments

Collective communication: Gather

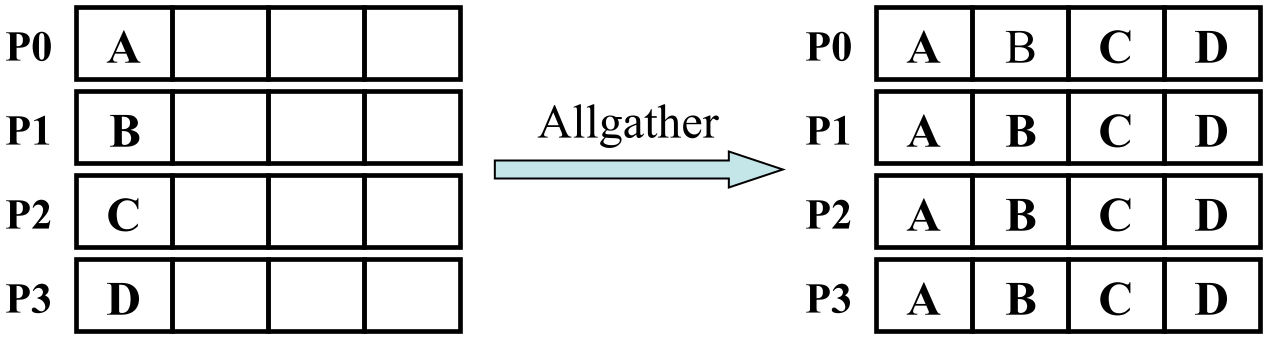
```
MPI_Gather(sendbuf,sendcount,sendtype,recvbuf,recvcount,
                          recvtype,root,comm,ierr)
```



- One root process collects data from all the other processes in the same communicator
- Must be called by all the processes in the communicator with the same arguments
- “sendcount” is the number of basic datatypes sent, not received (example above would be sendcount = 1)
- Make sure that you have enough space in your receiving buffer!

Collective communication: Gather to All

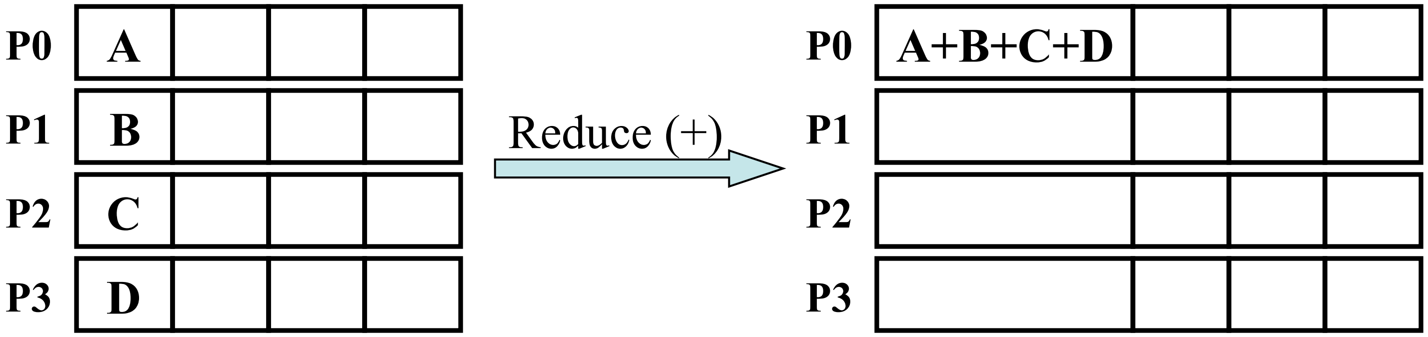
```
MPI_Allgather(sendbuf,sendcount,sendtype,recvbuf,recvcount,
              recvtype,comm,info)
```



- All processes within a communicator collect data from each other and end up with the same information
- Must be called by all the processes in the communicator with the same arguments
- Again, sendcount is the number of elements sent

Collective communication: Reduction

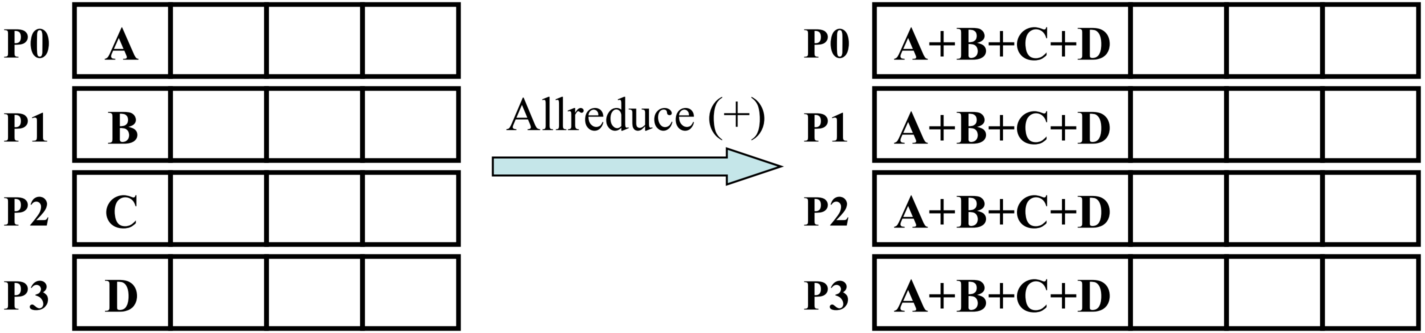
`MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm,ierr)`



- One root process collects data from all the other processes in the same communicator and performs an operation on the received data
- Called by all the processes with the same arguments
- Operations are: MPI_SUM, MPI_MIN, MPI_MAX, MPI_PROD, logical AND, OR, XOR, and a few more
- User can define own operation with MPI_Op_create()

Collective communication: Reduction to All

```
MPI_Allreduce(sendbuf,recvbuf,count,datatype,op,comm,ierr)
```



- All processes within a communicator collect data from all the other processes and performs an operation on the received data
- Called by all the processes with the same arguments
- Operations are the same as for MPI_Reduce

More MPI collective calls

One “root” process send a different piece of the data to each one of the other Processes (inverse of gather)

```
MPI_Scatter(sendbuf,sendcnt,sendtype,recvbuf,recvcnt,
            recvtype,root,comm,ierr)
```

Each process performs a scatter operation, sending a distinct message to all the processes in the group in order by index.

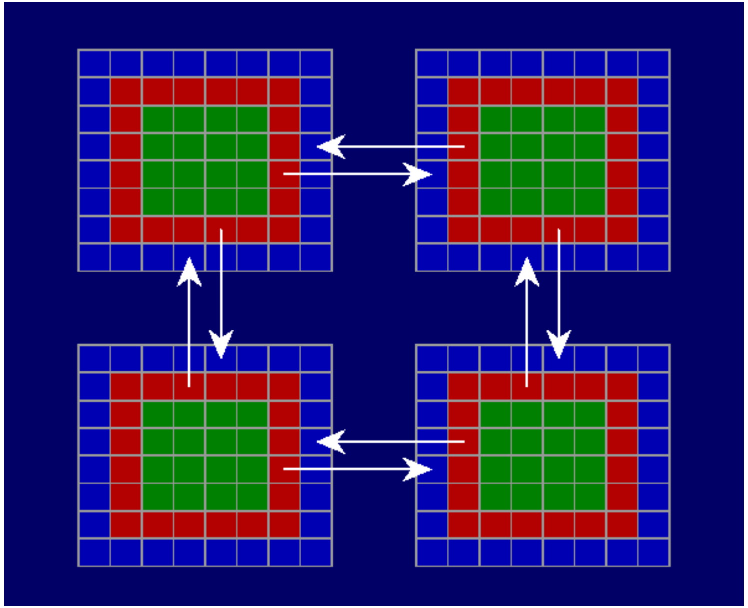
```
MPI_Alltoall(sendbuf,sendcount,sendtype,recvbuf,recvcnt,
             recvtype,comm,ierr)
```

Synchronization: When necessary, all the processes within a communicator can be forced to wait for each other although this operation can be expensive

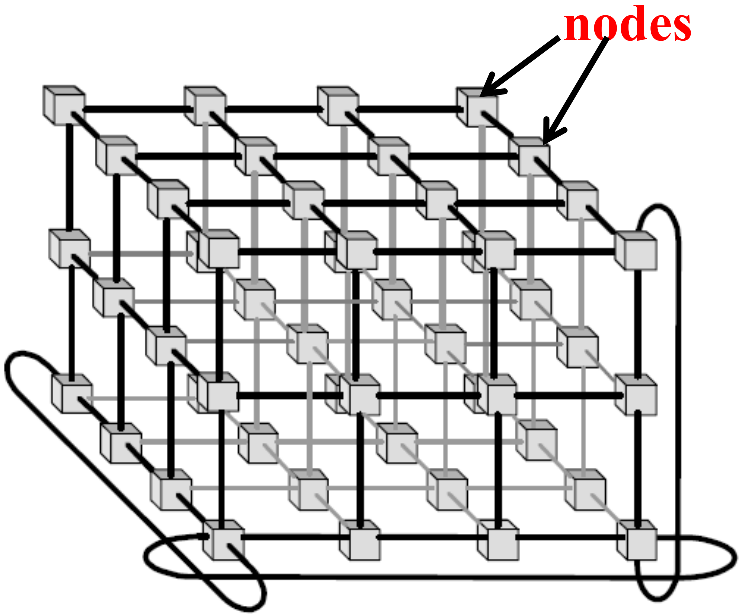
```
MPI_Barrier(comm,ierr)
```

MPI "topology" routines

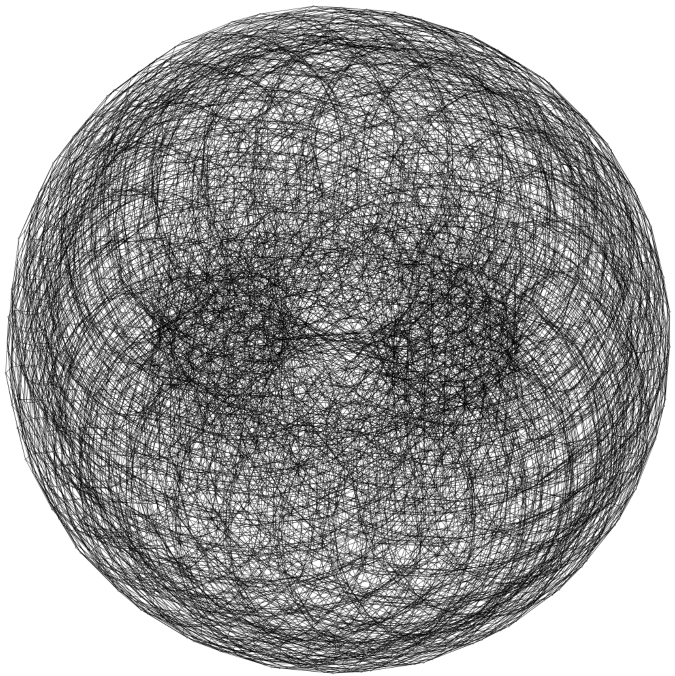
- MPI_Cart_create(MPI_Comm oldcomm, int ndim, int dims[], int qperiodic[], int qreorder,
  MPI_Comm *newcomm)
- Creates a new communicator **newcomm** from **oldcomm**, that represents an **ndim** dimensional mesh with sizes **dims**. The mesh is periodic in coordinate direction i if **qperiodic[i]** is true. The ranks in the new communicator are reordered (to better match the physical topology) if **qreorder** is true



Example of network topology



3D torus network interconnect
(e.g. Cray XE6 or XK7)



3D torus interconnect
On a large system!

MPI_Dims_create

- MPI_Dims_create(int nnodes, int ndim, int dims[])
- Fill in the **dims** array such that the product of **dims[i]** for i=0 to **ndim**-1 equals **nnodes**
- Any value of **dims[i]** that is 0 on input will be replaced; values that are > 0 will not be changed

MPI_Cart_create Example

- int periods[3] = {1,1,1};
  int dims[3] = {0,0,0}, wsize; MPI_Comm cartcomm;
- MPI_Comm_size(MPI_COMM_WORLD, &wsize);
  MPI_Dims_create(wsize, 3, dims);
  MPI_Cart_create(MPI_COMM_WORLD, 3, dims, periods, 1, cartcomm);
- Creates a new communicator **cartcomm** that "*may*" be efficiently mapped to the physical topology

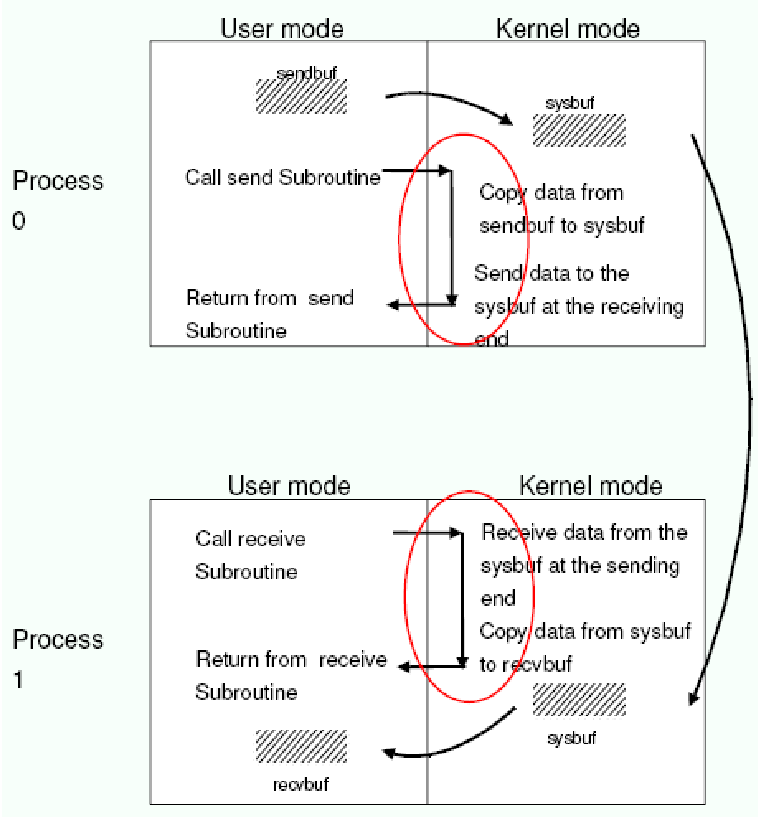
Determine Neighbor Ranks

- Can be computed from rank (in the cartcomm), dims, and periods, since ordering defined in MPI
- Easier to use either
  - MPI_Cart_coords
  - MPI_Cart_rank
  - MPI_Cart_shift

MPI_Cart_shift

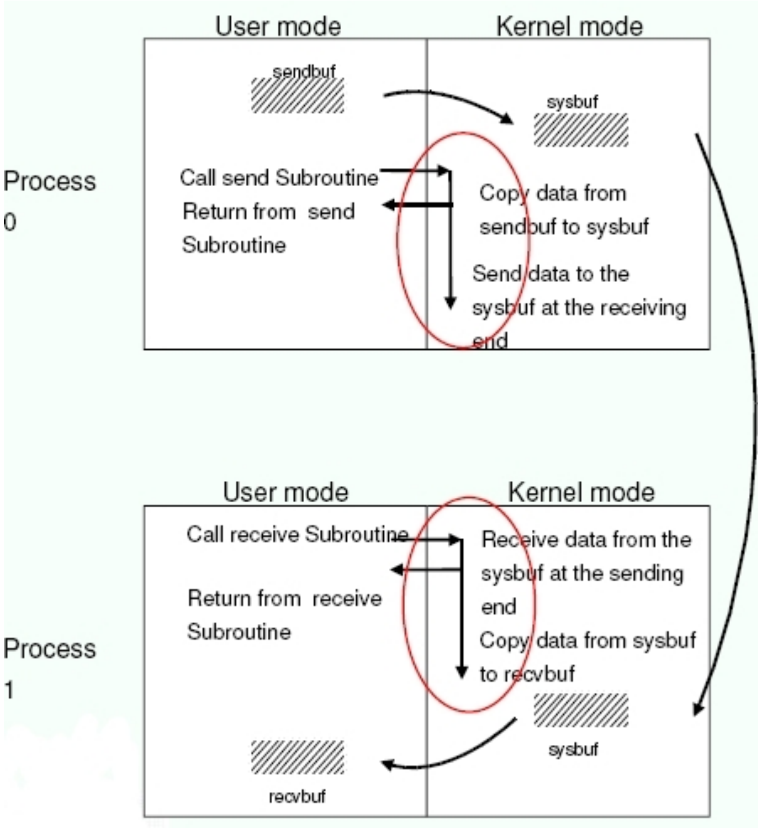
- MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)
- Returns the ranks of the processes that are a shift of **disp** steps in coordinate **direction**
- Useful for nearest neighbor communication in the coordinate directions
- Use MPI_Cart_coords, MPI_Cart_rank for more general patterns

Blocking communications



- The call waits until the data transfer is done
  - The sending process waits until all data are transferred to the system buffer (differences for *eager vs rendezvous* protocols...)
  - The receiving process waits until all data are transferred from the system buffer to the receive buffer
- All collective communications are blocking

Non-blocking



- Returns immediately after the data transferred is initiated
- Allows to overlap computation with communication
- Need to be careful though
  - When send and receive buffers are updated before the transfer is over, the result will be wrong

Non-blocking send and receive

Point to point:

```
MPI_Isend(buf,count,datatype,dest,tag,comm,request,ierr)

MPI_Irecv(buf,count,datatype,source,tag,comm,request,ierr)
```

The functions MPI_Wait and MPI_Test are used to complete a nonblocking communication

```
MPI_Wait(request,status,ierr)

MPI_Test(request,flag,status,ierr)
```

MPI_Wait returns when the operation identified by "request" is complete. This is a non-local operation.

MPI_Test returns "flag = true" if the operation identified by "request" is complete. Otherwise it returns "flag = false". This is a local operation.

MPI-3 standard introduces "non-blocking collective calls"

How to time your MPI code

- Several possibilities but MPI provides an easy to use function called "MPI_Wtime()". It returns the number of seconds since an arbitrary point of time in the past.

```
FORTRAN: double precision MPI_WTIME()
      C: double MPI_Wtime()
```

```
starttime=MPI_WTIME()
   ... program body ...
endtime=MPI_WTIME()
elapsetime=endtime-starttime
```

Debugging tips

Use "unbuffered" writes to do "printf-debugging" and always write out the process id:

```
C:        fprintf(stderr,"%d: …",myid,…);
Fortran:  write(0,*)myid,': …'
```

If the code detects an error and needs to terminate, use MPI_ABORT. The errorcode is returned to the calling environment so it can be any number.

```
C:        MPI_Abort(MPI_Comm comm, int errorcode);
Fortran:  call MPI_ABORT(comm, errorcode, ierr)
```

To detect a "NaN" (not a number):

```
C:        if (isnan(var))
Fortran:  if (var /= var)
```

Use a parallel debugger such as Totalview or DDT if available

References

- Just google "mpi", or "mpi standard", or "mpi tutorial"…
- http://www.mpi-forum.org (location of the MPI standard)
- http://www.llnl.gov/computing/tutorials/mpi/
- http://www.nersc.gov/nusers/help/tutorials/mpi/intro/
- http://www-unix.mcs.anl.gov/mpi/tutorial/gropp/talk.html
- http://www-unix.mcs.anl.gov/mpi/tutorial/

- MPI on Linux clusters:
  - MPICH (http://www-unix.mcs.anl.gov/mpi/mpich/)
  - Open MPI (http://www.open-mpi.org/)
- Books:
  - Using MPI "Portable Parallel Programming with the Message-Passing Interface" by William Gropp, Ewing Lusk, and Anthony Skjellum
  - Using MPI-2 "Advanced Features of the Message-Passing Interface"

Example: calculating π using numerical integration

```c
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    FILE *ifp;

    ifp = fopen("ex4.in","r");
    fscanf(ifp,"%d",&n);
    fclose(ifp);
    printf("number of intervals = %d\n",n);

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = 1; i <= n; i++) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;

    pi = mypi;
    printf("pi is approximately %.16f, Error is %.16f\n",
            pi, fabs(pi - PI25DT));
    return 0;
}
```

C version

```c
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i, j, tag, my_n;
    double PI25DT = 3.141592653589793238462643;
    double mypi,pi,h,sum,x,pi_frac,tt0,tt1,ttf;
    FILE *ifp;
    MPI_Status  Stat;
    MPI_Request request;

    n = 1;
    tag = 1;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    tt0 = MPI_Wtime();
    if (myid == 0) {
       ifp = fopen("ex4.in","r");
       fscanf(ifp,"%d",&n);
       fclose(ifp);
    }
 /* Global communication. Process 0 "broadcasts" n to all other processes */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Root reads input and broadcast to all

Each process calculates its section of the integral and adds up results with MPI_Reduce

```
…
    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid*n/numprocs+1; i <= (myid+1)*n/numprocs; i++) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;

    pi = 0.;  /* It is not necessary to set pi = 0 */

 /* Global reduction. All processes send their value of mypi to process 0
    and process 0 adds them up (MPI_SUM) */
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    ttf = MPI_Wtime();
    printf("myid=%d  pi is approximately %.16f, Error is %.16f  time = %10f\n",
           myid, pi, fabs(pi - PI25DT), (ttf-tt0));

    MPI_Finalize();
    return 0;
}
```

Python example

- http://mpi4py.scipy.org/docs/usrman/tutorial.html
- mpiexec -n 4 python script.py

Script.py

```
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()

if rank == 0:
    data = {'a': 7, 'b': 3.14}
    comm.send(data, dest=1, tag=11)
elif rank == 1:
    data = comm.recv(source=0, tag=11)
```

- Uses “pickle” module to get access to C-type contiguous memory buffer
- Evolving rapidly

Thank you...